Creating WinSystems ELS-2.0

There are literally hundreds of Linux distributions now available and just as many ways to create an embedded Linux system. Variety is one of the reasons Linux is so popular. Most users have their favorite distribution and prefer to work with that platform. These distributions, however, are often not practical for embedded systems. Typically embedded systems run from solid state memory and have limited resources. If you plan to run your system from a hard drive, then your favorite distribution should run on any WinSystems SBC, just as it would on an equivalent PC. But for the rest, they require a small Linux system which is reasonably easy to create and able to boot from solid state Flash drive.

As we were developing WinSystems Embedded Linux Server 2.0, we tried to document the steps taken to create the bootable image. Though we cannot cover every detail, it should provide a good starting point to develop your own embedded Linux system. For more detailed information, please refer to the published materials and websites referenced in "WinSystems ELS Overview. Our examples are based SimplyMEPIS-3.3, but the procedures should be very similar for most distributions.

This information is offered 'as is'. It is provided as an example only and is not intended for any particular application or purpose. WinSystems does not warrant its functionality and disclaims any liability resulting from its use. Additionally, WinSystems does not offer any Linux support services beyond assuring that the ELS is functional on the WinSystems single board computer on which it ships.

Tip: In the following examples, anything that is typed at a command prompt will be shown in **`Bold Courier`** *and items displayed on the terminal will be in plain* `Courier` .

Tip: If you purchased the ELS-2.0 Developer's Kit with the SDK4 hardware, SimplyMEPIS is already installed on the harddrive and updated with 'minicom' and 'docbook-utils.' The contents of the 'ELS-2.0 Resources CD' are located in the home directory for "admin." The accounts and passwords are ***admin:admin*** *and* ***root:root.***

1. First, we installed SimplyMEPIS-3.3 on our development workstation. SimplyMEPIS-3.3 is a Debian based distribution which comes on a single LiveCD. We find it to be very functional and easy to install. You can boot directly from the CD to test it on your system. Then follow the instructions to install to your hard drive. You can obtain SimplyMepis-3.3 at www.mepis.org. Please register your copy or sign up for the download subscription to support future development of this distribution. All examples below are based on this installation, though they can be easily adapted to most distributions.

2. Once we have Linux installed on our workstation, we want to login as a standard user and organize things a little. First, we create a directory to hold the ELS-2.0 project files. Then we create several subdirectories to better organize things. Please note, our example scripts expect files to be in certain subdirectories. So we recommend you go by our example, at least until you have time to edit the scripts appropriately.

   ```
   $ mkdir Els2.0
   $ mkdir Els2.0/sources Els2.0/rootfs Els2.0/scripts \
   > Els2.0/config-files Els2.0/images
   ```

3. Next, we set a variable to point to our main project directory. This variable is used throughout the build process. It is strongly recommended that you test this variable before executing any scripts or makefiles. Otherwise you may be putting files in places you really don't want them or overwriting important files that can cripple your new installation.

   ```
   $ export ELS=/home/george/Els2.0
   ```

   (You will need to replace '/home/george/Els2.0'with the path to the appropriate directory on your workstation.)

   ```
   $ echo $ELS
   /home/george/Els2.0
   ```

4. Since our ELS 2.0 needs to run on some slower processors, the kernel is built for the i486 processor. Our workstation, however, is a Pentium M 1.6 Ghz processor. We could cross-compile the entire system, but it's not necessary if the build system and target are x86 compatible. We will take a shortcut and trick the system into building for our i486. We chose to use a *uname* hack to accomplish this because of it's simplicity. We copy the original *uname* file to *uname.orig*. Then we create a new *uname* file that will search the original and replace the processor value with 'i486' (you may need a different value here depending on the target processor). Then we set the new file to be executable. Don't forget to restore *uname* to the orginal, once you have completed your work. *Tip: If you are using the target single board computer as your development platform, you can skip this section.*

```
$ uname -a
$ Linux GtLaptop 2.6.10 #1 Wed Feb 23 16:54:53 EST 2005 i686 GNU/Linux
$ su
Password:*****
# cd /bin
# mv uname uname.orig
# cat >/bin/uname <<"EOF "
>#!/bin/bash
>/bin/uname.orig "$@" | sed "s/i[456]86/i486/"
>EOF
# chmod 755 uname
# exit
$ uname -a
$ Linux GtLaptop 2.6.10 #1 Wed Feb 23 16:54:53 EST 2005 i486 GNU/Linux
```

5. Next,we get the necessary source files and place them in the $ELS/sources directory. We've listed the specific sources we used below along with the appropriate websites for download. The files are also included on the ELS-2.0 Resources CD.
   1. Linux kernel 2.6.10 (linux-2.6.10.tar.bz2) http://www.kernel.org/pub/linux/kernel/v2.6
   2. Hotplug scripts (hotplug-2004_09_23.tar.bz2) http://www.kernel.org/pub/linux/utils/kernel/hotplug
   3. Module tools (module-init-tools-3.1.tar.bz2) http://www.kernel.org/pub/linux/utils/kernel/
   4. Busybox 1.00 (busybox-1.00.tar.bz2) http://www.busybox.net/downloads
   5. Grub 0.95 (grub-0.95.tar.gz) ftp://alpha.gnu.org/gnu/grub
   6. Setserial 2.17 (setserial-2.17.tar.gz) http://sourceforge.net/projects/setserial
   7. Sysklogd (sysklogd-1.4.1.tar.gz) http://freshmeat.net/projects/sysklogd
   8. Kernel header patch (sysklog-1.4.1-kernel_headers-1.patch http://www.linuxfromscratch.org/patches/lfs/6.0/
   9. Signal handling patch (sysklogd-1.4.1-signal-1.patch http://www.linuxfromscratch.org/patches/lfs/6.0/
   10. Sysvinit 2.85 (sysvinit-2.85.tar.gz) http://freshmeat.net/projects/sysvinit
   11. Thttp HTTP Server (thttpd-2.25b) http://freshmeat.net/projects/thttpd
   12. VSFTPD FTP Server (vsftpd-2.0.3.tar.gz) http://freshmeat.net/projects/vsftpd
   13. ELS2.0 config-files and scripts http://developer.winsystems.com/Public/Linux
   14. PCI utilities (pciutils-2.1.11) http://www.kernel.org/pub/software/utils/pciutils

6. The system startup and configuration files used in the ELS-2.0 were downloaded above and need to be uncompressed. We will not go over all these files but they are based on a Debian style system. The *Linux* kernel configuration file and *BusyBox* configuration file are also included.

```
$ cd ${ELS}/config-files
$ cp ${ELS}/sources/els-2.0.1_config_ppm520.tgz ${ELS}/config-files
$ tar xvzf els-2.0.1_config_ppm520.tgz
```

7. The Linux kernel needs to be configured and built. To save a little space on our target, we built a compressed kernel image. The kernel configuration we used is provided *config-files* directory we extracted in step 6. You may want to change the kernel configuration to better match your application by running "make xconfig" prior to building the kernel image.

```
$ cd ${ELS}/sources
$ tar xvjf linux-2.6.10.tar.bz2
$ cd linux-2.6.10
$ cp ${ELS}/config-files/kernel-config/config-2.6.10_els-2.0_ppm520_20050617\
< .config
$ make clean
$ make bzImage
$ make modules
```

8. Since we might use several different kernel builds while testing our system, we use a separate directory to store our kernel images, modules, and configuration files. When we copy the kernel files to this directory we want to rename them so we can track different revisions.

```
$ cp arch/i386/boot/bzImage ${ELS}/images/bzImage-2.6.10-els-2.0
$ cp System.map ${ELS}/images/System.map-2.6.10-els-2.0
$ cp .config ${ELS}/images/config-2.6.10-els-2.0
$ make INSTALL_MOD_PATH=${ELS}/images/modules-2.6.10-els-2.0 modules_install
```

9. While we are dealing with kernel stuff, the *module-init-tools* utilities allow us to manage 2.6 kernel modules. They replace the *modprobe* tools used with the 2.2 and 2.4 kernels. Due to the a missing '*docbook-utils*' package, an error will be reported while creating the manual. This error can be safely ignored, since manuals only take up space on our target system.

```
$ cd ${ELS}/sources
$ tar xvjf module-init-tools-3.1.tar.bz2
$ cd module-init-tools-3.1
$ ./configure --prefix="${ELS}/rootfs" --enable-zlib
$ make
```

10. Though they may not apply to all embedded systems, the *hotplug scripts* allow the system to automatically recognize USB devices or PC-Cards that are plugged into a live system. This package provides the "helper "programs which are called by the kernel. First, let's decompress the scripts and enter the directory.
    1.

```
$ cd ${ELS}/sources
$ tar xvjf hotplug-2004_09_23.tar.bz2
$ cd hotplug-2004_09_23
```

    2. There is really nothing to build here but we do need to use an editor to set the prefix variable in the Makefile. This tells the installer where to put our scripts. Change the line as follows:

```
prefix=${ELS}/rootfs
```

11. The *Setserial* program is required to address several issues with serial ports. We use it to set the correct IRQ's for the 3rd and 4th serial ports on the single board computers. It is also necessary when interfacing with multi-port serial modules,such as the PCM-COM4A or PCM-COM8.

```
$ cd ${ELS}/sources
$ tar xvzf setserial-2.17.tar.gz
$ cd setserial-2.17
$ ./configure
$ make
```

12. BusyBox really is "The Swiss Army Knife of Embedded Linux ." It combines tiny versions of many common programs and utilities into a single small executable. Though the utilities in BusyBox generally have fewer options than their full-featured GNU counterparts, they are quite functional and behave very similar. BusyBox is used by many embedded Linux applications and distributions, as well as install/recovery disks for some major distributions. Busybox utilizes a menu based build system similar to the kernel menuconfig. Our configuration file is copied from the config_files directory to '.config' in our build directory. This will provide you with a functional starting point. You may want to try some of the other BusyBox features or disable functions not required.

```
$ cd ${ELS}/sources
$ tar xvjf busybox-1.00.tar.bz2
$ cd busybox-1.00
$ cp ${ELS}/config-files/busybox-config/busybox_config_els-2.0_20050422 .config
$ make menuconfig
$ make all
```

13. We chose *GRUB* as the system boot loader because it is extremely versatile and actively developed. We also feel it offers some advantages during development, when the command prompt can be very handy for testing configurations.

```
$ cd ${ELS}/sources
$ tar xvzf grub-0.95.tar.gz
$ cd grub-0.95
$ aclocal &&automake &&autoconf
$ ./configure --disable-ffs --disable-xfs --disable-jfs \
>--disable-vstafs --disable-reiserfs --disable-minix
$ make
```

14. Though *BusyBox* contains a system initialization utility, we chose to use the standard *Sysvinit* for our system. *Sysvinit* supports multiple runlevels and the standard inittab file. Please note, you will receive an error concerning the installation of the user manual when this program is installed. This can be ignored, since we do not care about the manual on our embedded system.

```
$ cd ${ELS}/sources
$ tar xvzf sysvinit-2.85.tar.gz
$ cd sysvinit-2.85/src
$ make
```

15. Again,we could have utilized the *syslog* and *klog* daemons provided with *Busybox* , but chose to go with the full versions for better configuration options. There are some issues with building sysklogd so we first need to patch the source. The patches are from www.linuxfromscratch.org. If you have not gone through a Linux from Scratch build, we highly recommend it. It can be very beneficial for learning how your Linux system fits together.
    1.
    ```
    $ cd ${ELS}/sources
    $ tar xvzf sysklogd-1.4.1.tar.gz
    $ cd sysklogd-1.4.1
    $ patch -Np1 -i ../sysklogd-1.4.1-kernel_headers-1.patch
    $ patch -Np1 -i ../sysklogd-1.4.1-signal-1.patch
    ```

    2. Once the source is patched, we need to edit the Makefile. The first change sets the installation directory to the correct directory. Then we comment out the line that would install the manual to our target. The new lines are shown here.
    ```
    BINDIR=${ELS}/rootfs/sbin
    #install: install_man install_exec (Manual is extra space)
    install: install_exec
    ```

    3. Finally,we can build the *sysklogd* source.
    ```
    $ make
    ```

16. Since this is an Embedded Linux Server, we now add some server capabilities. Installing a full Apache Web Server would defeat our purposes for a small system, so we chose the *Thttpd* HTTP server. *Thttpd* is a popular, small and very efficient HTTP server used on many websites.
    ```
    $ cd ${ELS}/sources
    $ tar xvzf thttpd-2.25b.tar.gz
    $ cd thttpd-2.25b
    $ ./configure
    $ make
    ```

17. Now we will add a popular FTP server, *vsftpd (Very Secure File Transport Protocol Daemon)*. This is a very popular FTP server and is even featured on *kernel.org* FTP site.
    ```
    $ cd ${ELS}/sources
    $ tar xvzf vsftpd-2.0.3.tar.gz
    $ cd vsftpd-2.0.3
    $ make
    ```

18. Before we build our system, let's extract the pciutils. These files can be useful when troubleshooting PCI devices. The files will be built and copied with our creation scripts
    ```
    $ cd ${ELS}/sources
    $ tar -xvzf pciutils-2.1.11.tar.gz
    ```

19. Going through the individual build tasks becomes repetitive so we created a few scripts to automate things. The scripts were downloaded above and are very simple. You should check the scripts to make sure they fit your system and workstation and edit them as needed.
    ```
    $ cd ${ELS}/scripts
    $ cp ${ELS}/sources/els-2.0.1_scripts_ppm520.tgz ${ELS}/scripts
    $ tar xvzf els-2.0.1_scripts_ppm520.tgz
    ```

20. Next, we create the root filesystem using the scripts. The '*make_els_rootfs*'script uses several other scripts to create the directories, devices, etc. You can review the individual scripts and edit them as needed for your system. You will need to have root privileges when executing these scripts.

```
$ cd $ELS/scripts
$ su
Password:*****
#./make_els_rootfs
```

Some errors will be reported due to missing manual files. These can be safely ignored, as long as those are the true errors.

21. Our basic root filesystem is now complete. Before we copy anything to the CompactFlash, we need to delete any current partitions and create a Linux partition. We use *fdisk* to re-partition the drive. Note, that these commands must be executed as root and we assume the CompactFlash is the first SCSI disk, since we use a USB-to-CompactFlash adapter.

```
$ su
Password:*****
# fdisk /dev/sda

Command (m for help): p

Disk /dev/sda: 260 MB, 260571136 bytes
16 heads, 32 sectors/track, 994 cylinders
Units = cylinders of 512 * 512 = 262144 bytes

Device Boot       Start       End         Blocks      Id    System
/dev/sda1             1       994         254448      83    Linux

Command (m for help):d
Selected partition 1

Command (m for help):p

Disk /dev/sda: 260 MB, 260571136 bytes
16 heads, 32 sectors/track, 994 cylinders
Units = cylinders of 512 * 512 = 262144 bytes

Device Boot       Start       End         Blocks      Id    System
/dev/sda1             1       994         254448      83    Linux

Command (m for help):n
Command action
e extended
p primary partition (1-4)
p
Partition number (1-4):
Value out of range.
Partition number (1-4):1
First cylinder (1-994,default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-994,default 994):
Using default value 994

Command (m for help):p
```

```
Disk /dev/sda: 260 MB, 260571136 bytes
16 heads, 32 sectors/track, 994 cylinders
Units = cylinders of 512 * 512 = 262144 bytes

Device Boot       Start       End         Blocks      Id    System
/dev/sda1             1       994         254448      83    Linux

Command (m for help):w
The partition table has been altered!
Calling ioctl()to re-read partition table.
Syncing disks.
```

22. Now that we have a valid Linux partition on the drive, we can create the filesystem on the CompactFlash card. We decided to use the standard Linux filesystem, ext2.

```
$ mke2fs -m0 /dev/sda1
$ tune2fs -c0 -i0 /dev/sda1
```

23. To copy the files to the CompactFlash, we first mount the device and then copy the files. You must have root privileges. We've used some short scripts here for convenience.

```
$ mkdir /mnt/cflash
$ ./m_cflash (mount CompactFlash on /dev/cflash)
$ ./cp2cflash (copies the files)
```

24. We need to install *GRUB* into the boot sector of the CompactFlash to make it bootable. *GRUB* is already installed on our workstation and we invoke it at the commandline. We are using a USB-to-CompactFlash Adapter, which appears as the first SCSI device. *GRUB,* however, recognizes it as the second hard drive. The *root* command sets the root filesystem for the installation. The *setup* command installs GRUB to the bootsector of the drive we specify. (CAUTION: You must be certain to specify the drive correctly. *GRUB* uses different naming conventions than Linux and a mistake could keep your workstation from booting. If in doubt, we recommend creating a *GRUB* boot floppy.)

```
$ grub
     GNU GRUB version 0.95 (640K lower / 3072K upper memory)

   [ Minimal BASH-like line editing is supported.  For the first word, TAB
     lists possible command completions. Anywhere else TAB lists the possible
     completions of a device/filename.]

   grub>root (hd1,0)
    Filesystem type is ext2fs,partition type 0x83

   grub>setup (hd1)
    Checking if "/boot/grub/stage1"exists...yes
    Checking if "/boot/grub/stage2"exists...yes
    Checking if "/boot/grub/e2fs_stage1_5"exists...yes
    Running "embed /boot/grub/e2fs_stage1_5 (hd1)"...16 sectors are embedded.
   succeeded
    Running "install /boot/grub/stage1 (hd1)(hd1)1+16 p (hd1,0)/boot/grub/stage2
   /boot/grub/menu.lst"...succeeded
   Done.

   grub>quit
```

25. Last we unmount the CompactFlash and give it a try!
    $ **./u_cflash** (unmounts the CompactFlash)

26. Remove the USB-CompactFlash adapter and install the programmed Cflash in the single board computer. Turn on the power and you should see a *GRUB* boot menu. After a short delay, the system should boot into your new Linux build.

OK, we have our embedded Linux system up and running. The config-files included setup the following users.

User	Password
root	root
admin	admin

Well now it is time to build your application programs. The distribution installed on your workstation has the basic editors and compilers. You can develop your applications and use *ftp* to transfer the files to the embedded system or setup a more complex scheme if you wish. You should have all the basics and most modern distributions have packages available to add functionality if something is missing.

One notable shortcut with our approach is using the GLIBC libraries from our workstation distribution. Our MEPIS libraries are already stripped and reasonably small, and we simply copied them into our filesystem. Ideally, the libraries should be built from scratch as well but this seemed unnecessary for our sample system. If you wish to use the smaller uCLIBC libraries or another version of the GLIBC libraries, the "LinuxFromScratch " website and "Building Embedded Linux Systems " book are excellent references. Setting up the authentication systems is often the most difficult part.

Though this document is not intended to be a comprehensive guide, hopefully it provides a good starting point for creating your embedded system.

Updates to this document and the files discussed can be found at:
http://developer.winsystems.com/Public/Linux

Revisions:

2006-01-31 – a) Added pciutils to source list.
b) Added step to extract pciutils.
c) Corrected various typographical mistakes and reworded a few statements.
d) Remove 'docbook-utils' requirement from Step 9 and documented error message.